CAMPUS
DE EXCELENCIA
INTERNACIONAL

Graduado en Ingeniería Informática

Universidad Politécnica de Madrid

Facultad de Informática

TRABAJO FIN DE GRADO

Diseño e implementación de un framework de funcionalidades Javascript en entornos de programación web

AUTOR: Álvaro José Aragoneses Martín

DIRECTOR: Ricardo Imbert Paredes

MADRID, JUNIO 2013

"*Pluralitas non est ponenda sine necessitate*"
(La pluralidad no se debe postular sin necesidad.)
Lex parsimoniae – Guillermo de Ockham

"Aprender a volar es todo un arte, aunque sólo hay que cogerle el truco:
Consiste en tirarse al suelo y fallar"
La Guía del Autoestopista Galáctico – Douglas Adams

Resumen

El presente documento aborda la problemática surgida en torno al desarrollo de una plataforma para gestionar las guías docentes de la Universidad Politécnica de Madrid, centrándose en el uso de las tecnologías Javascript, así como de lo algoritmos, plugins y bibliotecas auxiliares creadas y utilizadas. Por último, se muestran los resultados obtenidos del análisis y puesta en práctica de lo expuesto en el documento, así como conclusiones y sugerencias de futuras líneas de trabajo para este mismo proyecto.

Abstract

This document explains the problems found when developing a web service whose purpose is the management of learning guides at "Universidad Politécnica de Madrid". This final thesis focus on the use of Javascript technologies and the plugins, algorithms and auxiliar libraries used and developed. Finally, results of the analysis, development of the ideas exposed in this document, and conclusions and future working lines are presented.

Agradecimientos

A mis padres, Pedro y Mª Ángeles, por ayudarme a luchar por mi futuro.

A Jennifer, Javier, Salvador y muchas más personas indispensables en mi vida, por prestarme todo su apoyo y cariño en los buenos y malos momentos.

A Iñaki, Roberto, Eduardo y Carlos, fuentes de inspiración y ejemplo de pasión por su profesión; Y, por supuesto, a Hector y Sergio, grandes compañeros y laboratorio y grandes amigos, sin los que éste proyecto no habría sido posible.

Y, en especial, a aquellos que han dedicado, dedican o sueñan con dedicar su tiempo y su vida a la informática.

Índice de Contenidos

Índice de Fragmentos de Código

Índice de figuras

Parte I

Introducción a la plataforma

Capítulo 1

Introducción

En este capítulo se examinará con detenimiento la problemática en torno a las guías docentes, así como los trabajos previos y los objetivos fijados a la hora de desarrollar una nueva plataforma que dé soporte a esta herramienta fundamental en la docencia actual en la Universidad Politécnica de Madrid.

1.1. Estado de la Cuestión

A la hora de planificar una asignatura para su impartición se redacta un documento conocido como "guía docente". Dicho documento especifica todos los aspectos de dicha asignatura, tales como temario y contenidos, objetivos, competencias adquiridas, metodologías y técnicas docentes, cronograma, bibliografía y criterios de evaluación, entre otros. Este documento, si bien está sujeto a los cambios imprevistos dado que se trata de una planificación, ofrece tanto al docente como al alumno una visión a grandes rasgos de la asignatura, tanto en forma y contenido como en dedicación necesaria para superarla. Se trata de una herramienta fundamental en el desarrollo de los planes de estudios, ya que gracias a dichas guías puede obtenerse la lista de competencias adquiridas por un alumno a lo largo de su estancia en un determinado grado o máster.

Hasta hace poco, el sistema existente para el rellenado y gestión de las guías docentes venía haciéndose mediante diversos métodos: ficheros de texto, pdf's dinámicos, algún conato de plataforma para unificar la introducción de las guías, pero exclusivo para cada centro y, por tanto, atendiendo a sus particularidades, etc.

Por lo tanto, se impulsó desde el Vicerrectorado de Estructura Organizativa y Calidad la creación de un primer sistema, denominado "Europa", en el que, de manera centralizada y única, se introdujeran dichas guías de aprendizaje. Esta plataforma se planteó desde un principio como un flujo de trabajo o "workflow", en el que todo un paso debía estar correctamente cumplimentado para poder abordar el siguiente. Este hecho condicionó el sistema por completo, creando una serie de problemáticas tangenciales, como por ejemplo, la incapacidad de guardar parcialmente el contenido hasta no completar un paso, o la incapacidad, en versiones tempranas, de poder retroceder a pasos anteriores para modificar ciertos datos, funcionalidad que sí se añadió en posteriores revisiones, aunque presentaba la deficiencia de perder parcialmente los datos no guardados y, además, tener que repetir los pasos intermedios por los que se había retrocedido.

Todo este cúmulo de circunstancias derivaron en una sensación de frustración de los docentes, que tardaban una cantidad desmesurada de tiempo en cumplimentar sus guías

docentes. Tal era el problema que se decidió rehacer la plataforma, centrándose esta vez en las necesidades reales de los usuarios de la misma.

1.2. Motivación

El presente trabajo nace de la necesidad de ofrecer a los docentes una plataforma flexible y potente, adaptada a sus necesidades y escalable, centrada en la usabilidad y en la comodidad de sus usuarios para que cumplimenten de forma cómoda, rápida y eficaz sus guías docentes. Además, dicha plataforma formará parte del proceso de calidad promovido por el Vicerrectorado de Estructura Organizativa y Calidad y se integrará con el resto de herramientas creadas al amparo de dicho proceso.

1.2.1. Objetivos

- Crear una plataforma web capaz de gestionar todo el proceso de cumplimentación y aprobación de las guías docentes.
- Realizar dicha plataforma basándonos en las exigencias de los usuarios. Para ello, se realizarán entrevistas y test de usabilidad que nos ayuden a determinar las necesidades reales y los problemas potenciales de la plataforma a desarrollar.
- Realizar y documentar un diseño apropiado para la citada plataforma, de manera que sea escalable y fácilmente mantenible por los miembros de los futuros equipos responsables.

1.2.2. Alcance del presente trabajo

Tal y como indica su nombre, este trabajo se centra, sobre todo, en las tareas realizadas en torno al desarrollo de un marco de trabajo ("framework") en el lenguaje javascript, además de tratar de manera más tangencial otros lenguajes como PHP; además, este trabajo busca documentar los algoritmos y estructuras de programación utilizados en dicho lenguaje, que si bien no establecen un framework por si mismos, son una de las claves fundamentales de los buenos resultados obtenidos.

1.3. Estructura del trabajo

Tras lo visto hasta ahora, podemos definir la estructura del presente documento, que estará dividido en 4 partes:

- *Introducción a la plataforma*: Abarca los capítulos 1 y 2 del presente documento, y en ella se exponen tanto la introducción al problema como un somero análisis de la plataforma, así como de las tecnologías utilizadas.
- *Desarrollo de Funcionalidades*: Comprende los capítulos 3, 4 y 5, y alberga el análisis de los puntos más conflictivos del desarrollo de la plataforma, centrándose en los algoritmos y plugins utilizados, pero sin entrar en el detalle de las funciones auxiliares.

- *Herramientas Auxiliares para el Desarrollo*: Abarca los capítulos 6 y 7, y en ellos se contienen tanto la motivación y estructura de la biblioteca de funciones auxiliares creada durante el desarrollo de la plataforma como una serie de herramientas de generación de código para ayudar al desarrollador a la hora de crear nuevas funcionalidades o mantener las actuales, una herramienta para realizar consultas sobre la base de de datos o algunas pequeñas funcionalidades y metodologías para ayudar en la depuración del código.
- *Conclusiones*: En esta parte encontramos los dos últimos capítulos del trabajo: el capítulo 8 en el que se destacan los resultados globales de la aplicación de todo lo visto en el trabajo a la plataforma y, por último, el capítulo 9, que contiene unas breves conclusiones acerca del proyecto en su totalidad, especificando puntos fuertes y débiles de la plataforma y posibles vías de mejora.

Capítulo 2

Análisis de la plataforma

A continuación, una vez vista la problemática, pasaremos a comentar someramente tanto las tecnologías facilitadas para dar soporte a la plataforma, su estructura en forma de árbol de ficheros y los puntos fuertes y débiles que posee la misma, a fin de obtener unas conclusiones que nos ayuden a comprender el soporte y focalizar correctamente los esfuerzos.

2.1. Tecnologías utilizadas

La aplicación a construir está sustentada por un servidor (nombre-servidor) capaz ejecutar scripts escritos en el lenguaje PHP; para dotar de una mayor versatilidad, el servido cuenta con el motor de plantillas Smarty, capaz de proporcionar una abstracción del HTML generado y el PHP ejecutado, actuando de intermediario y evitando así que los ficheros PHP, destinados a funcionalidad, contengan sentencias que impriman sus resultados directamente sobre el navegador.

2.2. Estructura del árbol de directorios

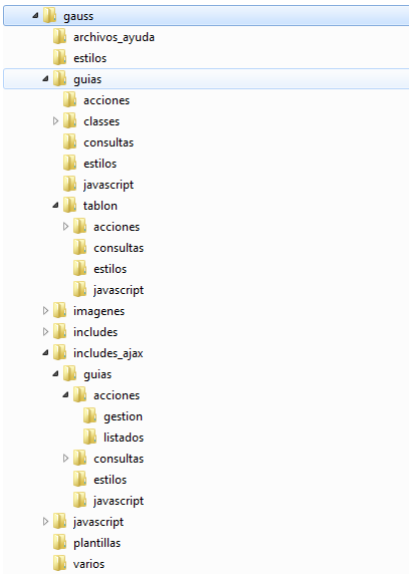

Figura 2.1: Estructura del árbol de directorios

2.3. Modelo de programación

Una vez visto el árbol de directorios, se aprecia que no se hace uso de ningún framework de los muchos disponibles en el mercado pero que, no obstante, se tiende a una estructuración basada en la ordenación lógica de la aplicación y sus ficheros.
Esta estructuración permite plantear la aplicación como un problema clásico de patrón Modelo-Vista-Controlador; en este caso, el modelo corresponderá a las clases de acceso a la base de datos, la vista a los ficheros de plantilla que Smarty es capaz de compilar y mostrar sobre el navegador y, por último, el controlador corresponderá al resto de ficheros de funcionalidad, escritos en PHP y Javascript.

2.3.1. Modelo de estructuración de ficheros en llamadas asíncronas

Mención aparte merecen los ficheros alojados en la carpeta "includes_ajax", en los que podemos encontrar una replica de parte del árbol de directorios; esto es así debido a que dicha carpeta es la única que contiene los permisos necesarios para responder correctamente a una llamada AJAX.

Adicionalmente, en el interior de la carpeta "includes_ajax/guias/acciones" encontramos dos carpetas, llamadas "gestionar" y "listados" que contienen, respectivamente, los ficheros que dotan de funcionalidad a las llamadas AJAX de los distintos ficheros y los ficheros que procesan y devuelven la información de la que se alimentarán los autocompletados de la plataforma (el sistema de autocompletado se explicará con mayor detalle en capítulos posteriores).

2.4. Conclusiones y futuras líneas de trabajo

A la vista del árbol de directorios y el modelo de programación, se puede observar que se trata de un intento de aplicar la metodología de los grandes frameworks a este sistema; sin embargo, inspirarse en dicha estructura no es suficiente para ofrecer toda la potencia de dichos frameworks, incluso mediante la inclusión de funciones de ayuda a la programación, ya que las funciones creadas para los citados marcos de trabajo, si bien no son tan específicas, están mucho más depuradas y su eficacia está fuera de toda discusión; por lo tanto, de propone como línea futura de trabajo la migración de todo el sistema a un framework de terceros, que ofrecería funcionalidades adicionales a las creadas *ex profeso* para la actual plataforma, como sistemas de seguridad o urls amigables.

Parte II

Desarrollo de Funcionalidades

Capítulo 3

Validación de formularios

Uno de los principales problemas a la hora de afrontar la programación de una plataforma que interactúe de forma intensiva con el usuario es la forma de validar los datos que dicho usuario introduce en los campos de los distintos formularios que ante él se le presentan; ésta interacción con el usuario ha de ser, por una parte, lo bastante flexible para no causarle frustración, lo suficientemente ágil para evitar esperas innecesarias y lo bastante robusta para evitar que usuarios malintencionados accedan a datos sensibles, así como que para que los usuarios que cometan un error no corrompan sus propios datos o los datos ajenos accidentalmente.

Es por ésto que el código requerido para la cumplimentación de dichos formularios ha ido evolucionando a lo largo de los años, al mismo tiempo que los usuarios se volvían más exigentes y las tecnologías avanzaban; actualmente es impensable un formulario en el que los datos deban validarse una vez se ha enviado el mismo, ya que genera una incertidumbre y frustración en el usuario que puede conllevar el abandono del uso de la aplicación. En su lugar, se ha establecido de manera tácita que el formulario debe ser capaz de validarse a si mismo antes de ser enviado al servidor, y en la medida de lo posible, en cuanto el usuario introduzca información o pierda el foco sobre el campo que esté rellenando.

Dado que la máxima de nuestra plataforma es la comodidad del usuario, creemos que una parte fundamental de nuestro desarrollo debe enfocarse al estudio e implementación de formularios agradables y cómodos para el usuario.

3.1. Estructura general de los formularios

A la hora de plantear los formularios, debemos analizar primero las tecnologías sobre las que se sustentan y su estructura. Tal y como se ha comentado anteriormente, el uso intensivo que la plataforma hace del framework Bootstrap nos obliga a seguir una determinada metodología, estructura y desarrollo de funcionalidad.

En el framework Bootstrap, los formularios están estructurados, al igual que otros elementos de la Web, en header, body y footer (cabecera, cuerpo y pié); el motivo de esta distinción es la de aplicar, automáticamente, estilos css en función del contexto en el que se encuentre el contenido, de forma mas o menos transparente al programador. En la sección header de un formulario bootstrap encontraremos, generalmente, el título de dicho formulario y una pequeña opción de cierre del formulario, que el propio framework mapea automáticamente a la acción correspondiente, dando así una alternativa al usua-

rio; por otro lado, en el footer se sitúan los botones de validación, envío, cancelación y otras acciones que afectan a la totalidad del formulario, y sobre los que se aplican estilos concretos en función del contexto de la acción (por ejemplo, los botones para formularios de confirmación de borrado se colorearán en rojo, mientras que los botones que confirman el envío de la información introducida por el usuario se colorean en azul, dejando el gris neutro para acciones reversibles, como cancelar el proceso).

Centrándonos en la tercera parte de un formulario, ésto es, el body o cuerpo, encontraremos que el framework Bootstrap aplica automáticamente estilos siguiendo una serie de reglas que el programador expresa mediante la utilización de capas html (" ") y clases que se le aplican a dichas capas. Esta mecánica abstrae al programador de conceptos como el "Responsive Web Design" ("Diseño Web Adaptativo", que se nutre de alterar los estilos CSS de una página para hacerla visible en, teóricamente, cualquier dispositivo) o la estructuración de los elementos de manera ordenada, aunque si precisa de reglas concretas para los estilos, debe implementarlas de forma manual, sobrescribiendo las reglas CSS del propio framework. Sin embargo, a efectos de funcionalidad, la imposición de esta estructura por parte del framework no entorpece la adición de funcionalidades, ya que en ningún momento impone la utilización de identificadores únicos a los campos del formulario, secciones que lo forman o el propio formulario en sí, dando al programador la opción de manejar estos identificadores a su elección. Adicionalmente, las clases añadidas por el programador a los distintos elementos del mismo son transparentes a Bootstrap, siempre que no entren en conflicto con las propias clases del framework, ofreciendo otra potente manera de acceder a los elementos albergados en los formularios.

3.2. Flujo de validación

Tanto o más importante que el aspecto visual de un formulario es el sistema que se utilice para comprobar que los datos aportados por el usuario cumplen los requisitos de forma para que la aplicación sea capaz de tratarlos de una forma segura; además, dicho sistema debe estar planteado de forma que permita una validación individual de cada campo, de todo el formulario y que, si alguno de los campos (o todos) son erróneos u obligatorios, no permita al usuario avanzar hasta no subsanar dicho error. La validación de todo el formulario, así como de cada campo, puede indicarse de manera explicita, con marcas y/o cambios en el color de la fuente del campo, o de manera implícita, mediante fórmulas menos intrusivas como la desactivación de los botones de envío del formulario o la utilización de mensajes de ayuda o "tooltips".

En nuestro caso, se decidió utilizar una mezcla de ambos métodos, indicando de manera explícita los campos erróneos, mediante el uso de las clases de error facilitadas por el framework Bootstrap, y deshabilitando el botón de envío del formulario, tanto en apariencia como en funcionalidad.

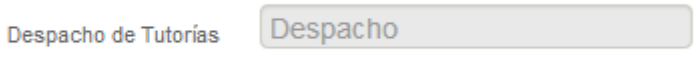

Figura 3.1: Ejemplo de un campo correcto

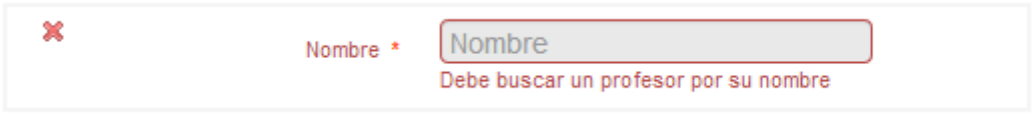

Figura 3.2: Ejemplo de un campo erróneo

Figura 3.3: Comparativa del aspecto del botón deshabilitado (izquierda) y habilitado (derecha)

Mención aparte merecen los mensajes de ayuda facilitados al usuario, los cuales se indican mediante el uso de "tooltips" flotantes que aparecen al situar el puntero del ratón sobre uno de los campos, y desaparecen al perder el foco sobre el mismo.

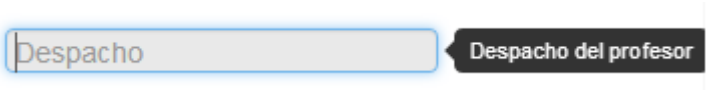

Figura 3.4: Ejemplo de un Tooltip

Además, en ciertos campos que requieran un formato especial de entrada, como pudieran ser fechas, rangos entre semanas, etc. se incluirá una descripción del formato como texto por defecto en el campo, funcionalidad facilitada por Bootstrap, además de repetir dicho texto debajo del propio campo, de forma que siempre quede visible para el usuario.

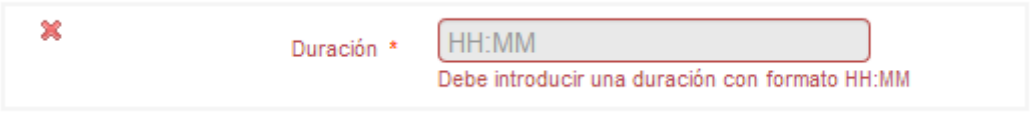

Figura 3.5: Ejemplo de un campo erróneo con descripción del formato a introducir

3.2.1. Estructura "Preparados, Apunten, Fuego"

A la hora de generar la funcionalidad de validación de los formularios, se hace indispensable establecer un flujo concreto de validaciones en cadena, de manera que cualquier mínimo cambio en alguno de los formularios desencadene los eventos que permitan determinar si el formulario en su totalidad es o no válido. Para lograr este efecto, nos basamos en una estructura repetitiva dividida en tres pasos, que nos permite controlar simultáneamente los cambios en todo el formulario y en cada uno de los campos individualmente; dicha estructura, a la que denominamos "Preparados, Apunten, Fuego" responde a la metáfora bélica de un pelotón que se prepara para realizar un ataque; en el momento en que el sargento ordena prepararse a los soldados, éstos preparan sus armas, que llevaban de forma cómoda para facilitar la marcha; tras la orden de apuntar, el pelotón carga y dirige los cañones hacia el objetivo; por último, la orden de fuego desencadena los disparos

hacia el objetivo.

De la misma forma, los formularios permanecen en un estado de reposo hasta que el usuario introduce algún cambio en un campo, momento en el que se comprueba si deben dispararse o no las funciones de validación; de ser así, el siguiente paso será la ejecución de una o varias funciones de validación en cascada, para determinar si el campo cumple con las condiciones de validación; de no ser así, se mostraría un mensaje con el error correspondiente y se marcaría ese campo como erróneo, abortando de este modo la cadena y desactivando, si no lo estaba ya, el botón de envío; en el caso de que se cumpla la condición de validación del campo, se evalúa si el formulario en su totalidad cumple con todos los requisitos, habilitando si procede el botón de envío.

Algoritmo: Preparados, Apunten, Fuego
Entrada: El usuario pierde el foco del campo.
1: $Condiciones[] \leftarrow$ Cadena de condiciones a cumplir.
2: $i \leftarrow 0$
3: $checkAllForm \leftarrow$ Función que comprueba todo el formulario.
4: **Para cada** $Condiciones[i]$ como $Actual$ **hacer**
5: **Si** $Actual \neq$ **Cierto entonces**
6: **Devolver** $Error$
7: Desactivamos el botón y marcamos el campo como erróneo.
8: **Fin Si**
9: **Fin Para**
10: **Si** $checkAllForm \neq$ **Cierto entonces**
11: **Devolver** $Error$
12: Desactivamos el botón del formulario.
13: **Si no**
14: **Devolver Cierto**
15: Activamos el botón del formulario.
16: **Fin Si**

Como podemos observar, este algoritmo deja bastantes decisiones a criterio del desarrollador, lo que ofrece una cierta flexibilidad para adaptarlo a situaciones similares, aunque las principales decisiones deberán tener en consideración factores como la capacidad de carga del servidor o la latencia de la red. La primera de estas decisiones recae en el propio disparador de las funciones de validación: un disparador demasiado sensible sobrecargará la ejecución de las validaciones, y ha de estar lo bastante afinado como para no generar errores de validación constantes, que frustrarían al usuario; por otro lado, un disparador demasiado tardío generaría frustración en el usuario por haber perdido hace tiempo el foco del campo, como podría ocurrir si validamos los campos al pulsar el botón de envío, al completar un conjunto de campos, etc. Para nuestra aplicación, decidimos que el disparador más equilibrado y que mejor se adecuaba a nuestra plataforma sería el evento "focus out", o pérdida del centro de atención (foco) sobre el campo a validar.

Sin embargo, existen otros disparadores, menos frecuentes, pero que también requieren de cierto estudio, como podría ser el evento "change", que detecta cualquier cambio en el campo en cuestión y que es útil para evaluar, en compañía de funciones que detectan

el valor seleccionado, un menú desplegable HTML (select); éste disparador genera una gran cantidad de información, y es útil para dotar a los campos de ciertos sistemas más elaborados que una comprobación (como, por ejemplo, sistemas de autocompletado, que veremos en un capítulo posterior).

Otro factor a considerar a la hora de evaluar que disparador utilizaremos es la cantidad de "falsos positivos" que es capaz de generar, es decir, las veces que se lanzará el evento sin que haya ocurrido ningún cambio aparente, o que se haya lanzado cuando el campo ha terminado en un estado idéntico al que tenía cuando el usuario centro el foco en él. Queda fuera del ámbito de este trabajo el estudio de funciones que miden la "suciedad" de un determinado campo, ya que, pese a ofrecer un mayor control sobre la cantidad de eventos lanzados y guardados en base de datos, su complejidad puede aumentar considerablemente en función del contexto en el que nos encontremos.

3.2.2. Problemática de las comparaciones: legibilidad y escalabilidad

Uno de los principales escollos a salvar en la validación de formularios es la cantidad de comparaciones, muchas de ellas iguales, que deben hacerse. Concretamente, nuestros formularios requisan de, al menos, una o más comparaciones por cada campo, a fin de detectar la idoneidad o no de los datos que posee, y una comparación adicional aplicada a todo el formulario, que determinará la activación, o no, del botón de envío. Contabilizando esto, tendríamos:

$$C = (x_1 + x_2 + \ldots + x_n) + (x_k)$$

Donde $x_1, x_2, \ldots x_n, x_k$ se refiere al número de comparaciones a realizar por cada campo. Normalmente, éstas comparaciones se reducirán a una por campo, simplificando el anterior conteo y convirtiéndolo en:

$$C = N + 1$$

Siendo N el número de campos.

Debido a ésto, la complejidad del código, así como su legibilidad, aumenta en función del número de campos y/o la interacción entre ellos, de manera que será más sencillo, a la hora de codificar, encontrar un formulario con campos disjuntos frente a uno con dependencias entre campos. Por cada uno de los campos dependientes, aumentaremos las comparaciones en función de la cantidad de ellos, por ejemplo, si tenemos 3 campos dependientes entre sí, deberemos codificar, en bruto, 9 comparaciones: por cada uno de los tres campos, la confirmación de ese campo y de su relación con cada uno de los demás. Contando con que sean dependencias simples, estaríamos hablando de una complejidad cuadrática, que implica una codificación en bruto de n^2 comparaciones, siendo n en número de campos dependientes, sumado al cálculo anterior.

A la vista de los datos, queda patente que un formulario básico genera una cantidad desmesurada de comparaciones duplicadas, las cuales son complicadas de mantener a largo plazo, con el añadido de que cada estructura es distinta, puesto que los campos interdependientes tendrán una validación en cascada (con estructuras $if, elseif, else$ encadenadas) mientras que la función de validación del formulario solo poseerá una rama if, pero que evaluará todas la condiciones anteriormente citadas, y que puede crear confusión a la hora de establecer correctamente el orden en que las comparaciones se ejecuten, así como los

conectores lógicos entre dichas comparaciones.

Antes de continuar, veamos un ejemplo real de la aplicación, para analizar detenidamente la problemática de la legibilidad:

```
1        // codigo previo...
2        if($.global.gga_pif_add_name.selected  &&
3            $.global.gga_pif_add_name.selected.label == $('#gga_pif_add_name').val() &&
4            !encontrarRepetidosPif('#gga_pif_add_name') &&
5            $(selector + '_mail').val().match('^[a-zA-Z_\.\+0-9]+@[a-zA-Z_0-9]+(\.[a-zA-Z
                 ]{2,6})+$') &&
6            $(selector + '_resp').val() != ''){
7                // Acciones si el codigo es valido
8        } else {
9                // Acciones si el codigo no es correcto
10       }
11       // Codigo posterior...
```

Listado 3.1: Ejemplo de código de validación

A la vista del código, observamos que la rama que evalúa el *if* general consiste en una sucesión de conectores lógicos de tipo *And*, de forma que aprovecharemos la "evaluación perezosa" [1] de Javascript, que sólo se presenta en los operadores "AND" y "OR". [1] Sin embargo, pese a que ésta práctica aumenta el rendimiento, dificulta enormemente la escalabilidad y la corrección de errores del formulario.

Para reducir este efecto, se propone como alternativa la creación de unas denominadas "cadenas de callbacks".

3.3. Cadenas de Callbacks

Denominaremos "Cadena de Callbacks" al uso de una estructura formada por funciones, o "eslabones", que se ejecuten en cascada y siguiendo un recorrido único, y que poseen la particularidad de tener, de manera explícita, una llamada al siguiente eslabón, o lo que es lo mismo, un "callback"[2]. La utilización de estos eslabones nos permite, por una parte, definir de manera estructurada y rápida un sistema de validación ordenado y legible, además de ofrecernos, de manera tangencial, una evaluación perezosa en el caso concreto tener comparaciones complejas.

La adopción de estas cadenas, además, permite la reutilización potencial del código, siempre que definamos una interfaz uniforme para las funciones de validación, y la facilidad para reescalar el formulario modificando una ínfima cantidad del mismo.

[1] Se denomina "Evaluación perezosa" a la propiedad de determinados lenguajes por la que solo evalúan las expresiones que consideran indispensables; por ejemplo, a la hora de evaluar una expresión lógica con conjunciones, interrumpirá la evaluación cuando uno de los elementos sea "false", ya que invalida el resto de la expresión.

[2] Llamada a una subrutina o función que es pasada como argumento a otra función, permitiendo grandes niveles de abstracción y reutilización de código

3.3.1. Los eslabones de la cadena: funciones envolventes y funciones de validación

Para garantizar que la cadena de callbacks cumple las propiedades anteriores, es necesario definir los parámetros de entrada y la funcionalidad básica de cada eslabón. Supongamos, para ello, una cadena enlazada en la que no existe un control general y es cada nodo el que, de forma autónoma, conoce su estado y es capaz de determinar si, al recorrerla, debe dar a conocer el siguiente eslabón o, por el contrario, detenerse en ella; para ello necesitará conocer, al menos, su estado interno y la referencia al siguiente eslabón. La determinación el siguiente eslabón se realizará pasando la función $x_n + 1$ a la función x_n, y añadiendo, al final de la misma, el siguiente código javascript:

```
1         if (callback && typeof(callback) === "function") {
2             callback();
3         }
```

Adicionalmente, cada eslabón debe ser capaz de realizar una validación completa, y en función de la misma, continuar el proceso o detenerlo y mostrar un error. Para ello, se propone que la validación en sí se realice en una función aparte, que reciba los parámetros necesarios y retorne verdadero o falso, en función de si es o no válido; el porqué de separar dicha función es poder reutilizarla mas adelante. Si la función retorna falso, será el eslabón el encargado de establecer los errores adecuados. El esquema de la función será, por tanto:

```
1     function eslabonNesimo(arg1, arg2..., callback){
2         // Comprobacion de la condicion
3         if(!validarEslabonNesimo(arg1, arg2...)){
4             // codigo para mostrar el error
5         } else {
6             // Llamada al Calback
7             if (callback && typeof(callback) === "function") {
8                 callback();
9             }
10        }
11    }
```

Como se puede observar, queda separada la funcionalidad especifica del mostrado de errores, que estará contenida en el propio eslabón, de las funcionalidades de validación, que podrán reutilizarse para la comprobación final del formulario en conjunto, para habilitar, o no, el botón de enviado del mismo.

3.3.2. Inicio y fin de la cadena: Disparadores y funciones de guardado

Una vez vista la estructura básica de la cadena, así como de sus eslabones, centrémonos ahora en el inicio y final de dicha cadena; como ya se ha visto anteriormente, el compromiso de disparador de funciones de validación al que se llegó tras el análisis de la estructura "Preparados, apunten, fuego" determina que el disparador idóneo para las validaciones en nuestro caso particular será la pérdida de foco; gracias a las funciones que ofrece el framework JQuery, podemos obtener un escuchador de dicho evento y enlazarlo con el uso de una de las funciones de validación (o una cadena de Callbacks, según proceda), de la forma:

```
1     $(selector).on('focusout', function(event){
2         // Llamada a la/s funcion/es correspondiente/s
3     });
```

El hecho de encerrar la llamada a la validación en una función anónima responde al hecho de poder capturar, de manera transparente a la función de validación, el objeto que contiene la información del evento, de manera que podamos obtener el identificador único del objeto que disparó dicho evento (llamado "selector" en el código, y al que se le ha asociado el escuchador del evento) y controlar la propagación del mismo, así como el comportamiento por defecto de determinados elementos que puedan recibir el evento.

El final de una cadena de callbacks no entraña más complejidad salvo por el detalle de que debe ser éste último paso el que valide por completo el formulario y active, si procede, el botón de envío. En caso de no validar el formulario, además de desactivar visualmente el botón cambiará la funcionalidad de envío del formulario por la función que valida dicho formulario en su conjunto; de esta manera, solventamos pequeños problemas, como el que un usuario introduzca un dato erróneo, lo valide más tarde y, en lugar de primero pulsar en cualquier parte del formulario, activando el evento focusout, para después pulsar el botón, se permita que dicho usuario pueda pulsar sobre el botón directamente y desencadenar la misma funcionalidad; adicionalmente, se debe asignar la misma función de validación al evento "onmouseover", para que el usuario obtenga retroalimentación visual al presentársele el problema anteriormente citado, ya que el mero hecho de posicionar el cursor sobre el botón valida el formulario en su totalidad.

3.3.3. Problemática de las Cadenas de Callbacks

El principal problema que presentan las cadenas de Callbacks es su uso tan intensivo de funciones, que pueden generar usa saturación del heap y, por tanto, disminuir la eficiencia del código; el uso de éste sistema no está recomendado para formularios cuyos campos solo precisen una validación y no exista la posibilidad futura de aumentar las comprobaciones sobre dicho campo. Adicionalmente, una mala estructuración del código puede acarrear molestos problemas de legibilidad y mantenimiento, si bien es sencillo entender la estructura, se aconseja evitar en la medida de lo posible las expresiones ofuscadas que dificulten el entendimiento de las funciones, sobre todo las de validación, siendo éstas las más propensas a errar y las más difíciles de depurar en el caso de no estar adecuadamente estructuradas.

3.4. Conclusiones y Futuras Líneas de Trabajo

Se propone, a la vista de lo anteriormente expuesto, las siguientes líneas de trabajo futuras:

- **Serializador de formularios:** Un problema asociado al sistema de validación de los formularios es la gestión de los datos contenidos en el mismo, la cual puede ser tediosa y generar errores. Se propone la creación de un sistema que, en base a clases CSS sobre el HTML, serialice de manera automática el contenido de un formulario, una vez validado, y lo concatene en una cadena de texto que se autoevaluará en el lado del servidor al ser enviada por POST.
- **Array de condiciones y Autoevaluador:** Tal y como se ha visto, las funciones de validación poseen una interfaz simple e verdadero y falso, lo que las convierte en fácilmente automatizables; se propone, por tanto, la creación de una función que

valide un array de funciones, encapsulando toda la funcionalidad de la función de evaluación del formulario en su conjunto y creando un nivel adicional de abstracción para el desarrollador, el cual ya no deberá preocuparse de mostrar los errores y activar o desactivar el botón de envío; en necesario para que dicha funcionalidad trabaje correctamente el establecimiento de una serie de prefijos y sufijos sobre los identificadores de cada campo que faciliten la tarea de validar automáticamente el formulario, como pudiera ser, por ejemplo, la adición del sufijo "_err" en el div que contiene el mensaje de error o la concatenación del sufijo "_div" para referirse al recuadro que envuelve el campo en caso de error.

Capítulo 4

Autocompletado

Actualmente, los sistemas de autocompletado en los campos de introducción de texto son una de las herramientas más utilizadas y, al mismo tiempo, más complejas de implementar, pese a que su uso se ha generalizado, en gran medida gracias a los sistemas de búsqueda de páginas web, presentes en plataformas como Twitter, Facebook, Wolfram Alpha o el gran impulsor de estos sistemas, Google. Tal es su extensión que muchos usuarios ven los autocompletados como algo esencial a la hora de realizar búsquedas e introducir texto en campos, debido, principalmente, a que los autocompletados solventan, muchas veces, el problema del formato en el que debe introducirse el texto en un campo, facilitando el trabajo en ambos lados de la aplicación, ya que, por una parte, el usuario se despreocupa de introducir el formato correcto, puesto que ya lo hará el autocompletado por él, y por otra, el número de comprobaciones de seguridad del lado del servidor que debe implementar el desarrollador decrementa enormemente, debiendo unicamente comprobar que lo introducido se ajusta al patrón que necesita, e incluso invalidando el envío de la información al servidor desde el lado del cliente (para más información sobre la problemática de las validaciones de usuario, véase el capítulo 3).

Sin embargo, como ya se ha dicho, se trata de unas herramientas de gran complejidad, y requieren de una orquestación entre la parte del cliente y del servidor para funcionar correctamente, además de presentar otras muchas particularidades que se comentan con mayor detalle a continuación.

4.1. Planteamiento del problema

Para que tanto el usuario, a la hora de escribir, como el desarrollador, a la hora de recibir lo escrito, puedan beneficiarse de las ventajas de los sistemas de autocompletado, deben cumplirse una serie de reglas, a saber:

1. El autocompletado debe poseer la información del contexto en el que se encuentra; en caso de tener más información, debe distinguirse claramente. En caso de no poseer la información del contexto, debe recuperarla de forma transparente al usuario y en la menor cantidad de tiempo posible.

2. El autocompletado debe proveer un sistema eficaz de corrección de errores.

3. El autocompletado debe permitir al usuario introducir cualquier cadena de texto y, en caso de que dicha cadena concuerde con una de las posibles opciones del autocompletado, debe detectar este hecho automáticamente y ajustar, de manera transparente al usuario, la información al formato requerido.

Esta imposición de funcionamiento crea una serie de puntos conflictivos a la hora de codificar el autocompletado, ya que muchos de los sistemas de autocompletado existentes cubren solo parte de estos puntos, siendo la recuperación de información bajo demanda la parte más común que se cede al programador, debido a que cada lógica de programación es distinta.

4.2. Posibles soluciones

A continuación, se muestra un breve lista de los sistemas que se evaluaron como posibles herramientas a utilizar, así como sus pros y contras.

4.2.1. AutoComplete 1.2 Scriptaculous

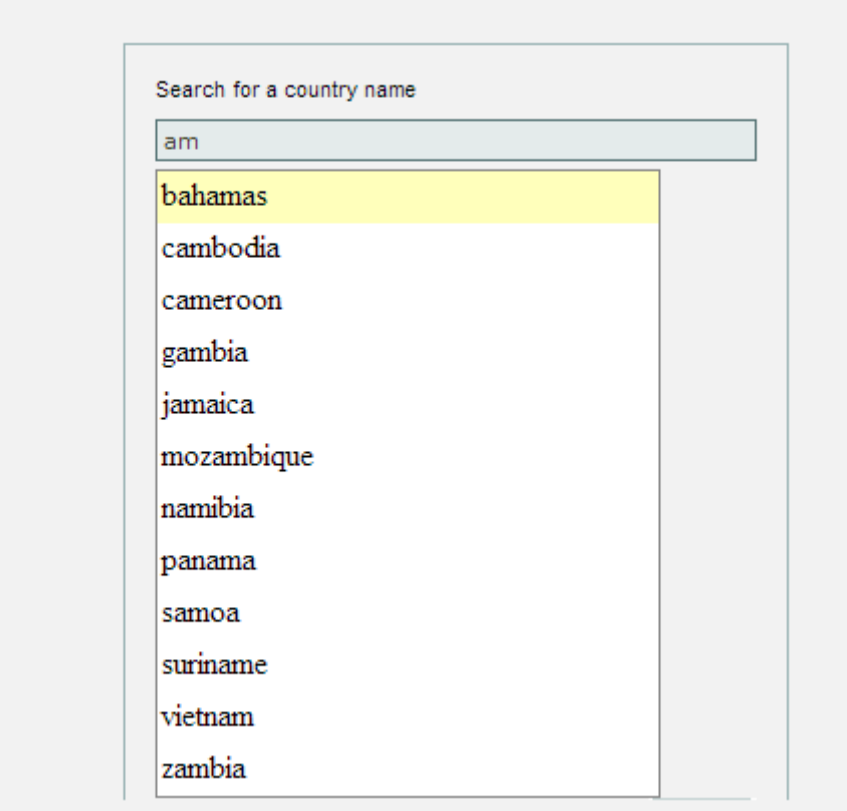

Figura 4.1: Ejemplo del plugin Scriptaculous Autocomplete

Basado el el framework "prototype.js", ofrece un autocompletado en forma de lista desplegable, con una animación fluida y elegante.
Pros: Fácil de usar, ofrece de manera nativa la obtención de datos remotos.
Contras: Requiere la utilización de "Prototype.js", lo que puede ocasionar conflictos con el framework que venimos utilizando en la plataforma, "Jquery", ya no a nivel de funcionalidad sino en el estilo de codificación y las funcionalidades ofrecidas.

4.2.2. FLEXBOX Autocomplete

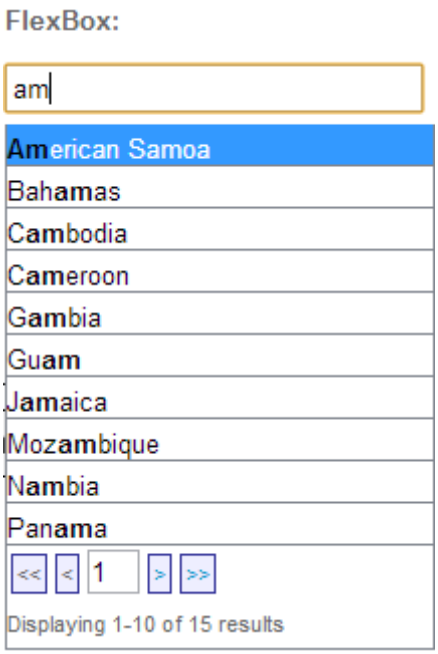

Figura 4.2: Ejemplo del plugin FLEXBOX Autocomplete

Plugin funcional sobre el framework Jquery, provee una amplia variedad de opciones de configuración, soporte nativo para fuentes remotas e incluso la capacidad de elegir el aspecto de la flecha desplegable.
Pros: Fácil y muy bien integrado con Jquery.
Contras: Requiere del uso de una estructura específica de ficheros.

4.2.3. JqueryUI Autocomplete

Figura 4.3: Ejemplo del plugin Jquery Autocomplete

Uno de los plugins mejor documentados y con un soporte más refinado, permite mucha flexibilidad a costa de aumentar su complejidad, pero su funcionamiento básico es impecable. Además, al tratarse de un plugin oficial de JQuery, se integra a la perfección con dicho framework.
Pros: Fácilmente ampliable, buena API, soporte oficial y gran comunidad de usuarios.
Contras: Cualquier personalización fuera de su funcionalidad original exige un conocimiento extenso del funcionamiento interno del plugin, o en su defecto, la utilización de plugins de terceros sobre éste plugin.

4.2.4. Elección de la tecnología a utilizar

A la vista del anterior análisis, se decidió elegir el plugin "JQueryUI Autocomplete" para realizar los sistemas de autocompletado. Para ello, se creo en un fichero común de Javascript dos funciones, llamadas "AddAutocomplete" y "AddRemoteAutocomplete" que reciben, entre otros parámetros, el campo de texto sobre el que aplicarle el plugin.

4.3. Adaptación de la solución

A continuación, se muestra el proceso seguido para la adaptación de la solución elegida a nuestra problemática particular

4.3.1. Problemática de la adaptación de la solución elegida

Uno de lo principales puntos a tratar en nuestra aplicación, una vez escogido el sistema de autocompletado a utilizar, es el proceso de adaptación a nuestra plataforma, atendiendo, principalmente, a aspectos como la funcionalidad y la visualización de los datos.

A este respecto, se decidió crear las dos funciones envolventes antes citadas, "AddAutocomplete" y "AddRemoteAutocomplete" para enmascarar los detalles del tratamiento de la fuente de los datos, así como la visualización, que se decidió homogeneizar de forma que cada una de las opciones seleccionables en el listado del autocompletar posea, además de una etiqueta identificativa, una pequeña línea con información adicional, que variará en función del contexto, por ejemplo, en el caso de un autocompletado de profesores por su nombre, cada opción mostrará, debajo del nombre, su dirección de correo electrónico.

Por otra parte, la funcionalidad fue desde el principio una prueba de ingenio, ya que no solo debíamos mostrar la información, sino además ser capaces de mostrarla en orden en función de la cadena introducida; por ejemplo, si el usuario intenta buscar las ocurrencias de la cadena "ca", se debe mostrar la cadena "calculo" como primera ocurrencia; esta funcionalidad debe tratarse de forma concreta, ya que, por defecto, el autocompletado de Jquery devuelve todas las cadenas que contienen como subcadena la indicada, pero de manera desordenada, por lo que puede aparecer como primera ocurrencia la cadena "al" la palabra "calculo" cuando debería ser "algebra".

Por otro lado, la fuente de la que se obtienen los datos procede casi siempre de una llamada AJAX al servidor, por lo que debíamos crear una interfaz uniforme para todas las llamadas de este tipo; finalmente, se recurrió a la creación de una serie de ficheros php que generaran un array de objetos javascript, los cuales poseerían toda la información necesaria, pero obligatoriamente poseerían dos campos, llamados "label" y "value", indispensables para crear el desplegable adecuadamente. Además, se requería que una vez el usuario seleccionara un dato, se pudiera extraer información de éste, por lo que se decidió crear una copia del objeto en una variable global, llamada "$.global.<namespace>.selected", en la que se almacenaría, de forma global pero delimitada por un espacio de nombres, el contenido del objeto. Las particularidades de la creación de espacios de nombres y de la copia de objetos en javascrpit se detallarán en el capitulo 6.

4.3.2. Código de ejemplo

A continuación, se muestra en detalle el código fuente de la función "AddRemoteAutocomplete", así como un ejemplo de uso.

```
1  function addRemoteAutocomplete( input, url, post, descriptionDiv, callback){
2          post = defaultValue(post, "");
3          descriptionDiv = defaultValue(descriptionDiv, "");
4          // Obtenemos un nivel de indireccion
5          var indirect = input.substring(1);
6          $.global = $.global || {};
7          eval ("$.global." + indirect + " = {}; var almacen = $.global." + indirect + ";");
8          almacen.objs = [];
9
10         var poi = $.getJSON(url,post)
11                 .fail(function(data, textStatus, jqXHR){
12                     console.error(textStatus);
13                     console.error(data.responseText);
14                 })
15                 .done(function(data) {
16                     try{
```

```
17                      almacen.objs = eval(data);
18                      // normalizamos texto
19                      for (d in almacen.objs){
20                          almacen.objs[d].label = htmlentitiesToUTF8(almacen.objs[d].
                                label);
21                      }
22                      if (callback && typeof(callback) === "function") {
23                          callback();
24                      }
25                  }catch(e){
26                      console.error(data);
27                  }
28              });
29
30          var ret = $(input).autocomplete({
31              minLength: 0,
32              messages: {
33                  noResults: null,
34                  results: function() {}
35              },
36              // obtenemos los datos de una fuente remota
37              source: function(request, response) {
38                      var term = $.ui.autocomplete.escapeRegex(request.term)
39                      , startsWithMatcher = new RegExp("^" + term, "i")
40                      , startsWith = $.grep(almacen.objs, function(value) {
41                          value = value.label || value.value || value;
42                          return startsWithMatcher.test(value);
43                      })
44                      , containsMatcher = new RegExp(term, "i")
45                      , contains = $.grep(almacen.objs, function (value) {
46                          value = value.label || value.value || value;
47                          return $.inArray(value, startsWith) < 0 &&
48                              containsMatcher.test(value);
49                          });
50                      for(var i = 0; i < startsWith.length; i++){
51                          var j = inArraySelectableObjs(startsWith[i], contains);
52                          if ( j != -1){
53                              contains.splice(j,1);
54                          }
55                      }
56                      response(startsWith.concat(contains));
57              },
58              focus: function(event, ui) {
59                  $(input).val(ui.item.label);
60                  almacen.selected = new clone(ui.item);
61                  return false;
62              },
63              select: function(event, ui) {
64                  $(input).val(ui.item.label);
65                  almacen.selected = new clone(ui.item);
66                  return false;
67              },
68              error: function(event, ui) {
69                  console.error('ERROR en AUTOCOMPLETAR');
70              }
71          });
72          $('.ui-helper-hidden-accessible').css('display', 'none');
73
```

```
74          ret.data( "autocomplete" )._renderMenu= function( ul, items ) {
75                  var that = this,
76                  currentCategory = "";
77                  if (items.length > 0 ){
78                      $.each( items, function( index, item ) {
79                      if ( item.category != currentCategory ) {
80                          ul.append( "<li class='ui-autocomplete-category'>" + item.category
                                + "</li>" );
81                          currentCategory = item.category;
82                      }
83                      that._renderItem( ul, item );
84                    });
85                  } else {
86                      ul.append( "<li class='ui-autocomplete-category'>No se encontraron
                            resultados</li>" );
87                      that._renderItem( ul, item );
88                  }
89              }
90
91          ret.data( "autocomplete" )._renderItem = function( ul, item ) {
92              return $( "<li>" )
93                  .data( "item.autocomplete", item )
94                  .append( "<a>" + item.label + "<br>" + item.value + "</a>" )
95                  .appendTo( ul );
96          }
97          return ret.data( "autocomplete" );
98      }
99      // Ejemplo de uso:
100     addRemoteAutocomplete(
101             "#idAutocompletado",
102             '<ruta/al/listado.php>',
103             {
104                 post_var1: var1,
105                 post_var2: var2
106             }
107         );
```

Listado 4.1: Definición y ejemplo de uso de la función "addRemoteAutocomplete"

4.4. Conclusiones y Futuras Líneas de Trabajo

Tras el análisis mostrado en todo el capítulo, podemos concluir que los sistemas de autocompletado son piezas complejas de construir y mantener, pero que aportan enormes ventajas a los usuarios. Tal y como se ha podido ver, el código fuente, pese a ser complejo, es pequeño y relativamente fácil de encapsular a una función, por lo que es muy factible aislarlo en una biblioteca de funciones, con todas las ventajas que ello conlleva (no replicación de código, facilidad de depuración, etc.).

Como línea futura de trabajo, se propone una modificación al sistema de manera que sea capaz de recibir objetos javascript no limitados por los campos "label" y "value", y con la posibilidad de realizar búsquedas en función de cualquiera de los campos de dicho objeto, todo en un mismo autocompletado.

Capítulo 5

Tablas reordenables

Durante este capítulo trataremos de abordar la problemática que se ha dado en torno a la construcción de tablas reordenables, con filtros dinámicos y petición de información mediante llamadas AJAX.

5.1. Problemática a resolver

Las tablas son, en general, un mecanismo de gran potencia a la hora de representar datos ordenados en forma matricial, permitiendo al usuario hacer una comparativa de una o varias filas con las demás, en función de los campos representados en las columnas. Este hecho ha sido utilizado para representar en nuestra plataforma listados de información muy similar, facilitando su análisis a los usuarios, como puede ser, por ejemplo, en el caso de aprobación de asignaturas, que se representa en forma de tabla donde cada fila representa una asignatura y cada columna una de las propiedades que, en conjunto, permiten diferenciar a dicha asignatura de las demás.

La problemática, siguiendo el ejemplo arriba expuesto, viene a la hora de tratar de filtrar o reordenar las filas en función de distintos parámetros, como planes de estudios o nombre de las asignaturas, problema que requiere de la utilización de métodos "Javascript" para dotar a la tabla de las siguientes funcionalidades:

- Reordenado de la tabla en función de un campo, tanto de forma ascendente como descendente.
- filtrado de las filas en función de un campo, haciendo desaparecer aquellas que no cumplan dicho filtro.
- capacidad de añadir o eliminar filas de manera dinámica.
- Capacidad de expandir una fila, de forma que se pueda obtener más información que la ofrecida por los campos de la tabla para el contexto de la información ofrecida en esa fila.

5.2. Solución adoptada: Jquery Datatables

5.2.1. Funcionamiento básico del Plugin Jquery Datatables

El plugin para el framework Jquery provee la funcionalidad necesaria para el reordenado dinámico de la tabla, en función de una de sus columnas, de manera que, haciendo click

sobre el nombre de dicha columna, toda la tabla se reordenará automáticamente siguiendo un orden alfabético descendente (de la "A" a la "Z"); si volvemos a hacer click en la columna, el orden cambiará a ascendente (de la "Z" a la "A"). Sucesivos clicks alternarán entre estas dos ordenaciones.

Figura 5.1: Ejemplo de una tabla reordenable

5.2.2. Filtrado por columnas: Sorting Plugin

Tal y como e ha visto en la sección anterior, el jquery, y en especial, su plugin Datatables ofrecen gran parte de la funcionalidad requerida para nuestro proyecto; sin embargo, el filtrado por columnas, es decir, hacer desaparecer las filas que no cumplan la condición de tener el valor requerido en una de sus columnas, no está soportado de forma nativa, y para ello se ha de utilizar un plugin llamado "Sorting plugin".

Sin embargo, es necesario realizar una serie de modificaciones al código existente para adecuarlo visualmente a nuestras necesidades.
El primer punto que el plugin requiere es la utilización de un footer en la tabla, en el que se especificará el numero de filtros y un mensaje previo para ellos; al quedar en la parte inferior de la tabla, dichos filtros son poco visibles y decidimos ocultarlos por medio de Jvascript.
El siguiente punto a tratar es el tipo y contenido de los filtros, además de su nueva ubicación; para ello, nos valimos de las funcionalidades dadas por el propio plugin, en el que ofrece varios tipos de filtrado, de entre los que elegimos dos:

- Una lista de elementos, en los casos en los que se presentaran pocas opciones (menos de 7 o 10) como pueden ser la lista de todos los planes de estudios presentes en la plataforma.

- Un campo de texto que posee su propio autocompletado, de manera que se consigue filtrar por campos más variados, tales como el nombre de una asignatura.

Por último, el plugin nos permite situar el campo de filtrado allá donde queramos, eso si, siempre como una copia del que queda bajo la tabla (de ahí la necesidad de ocultar esos campos); en nuestro caso, elegimos situar los filtros dentro de una clase retractable, de manera que pueden ocultarse visualmente en caso de ser necesario.

Figura 5.2: Ejemplo de tabla con los filtros desactivados

Figura 5.3: Ejemplo de tabla filtrando los resultados según su plan de estudios

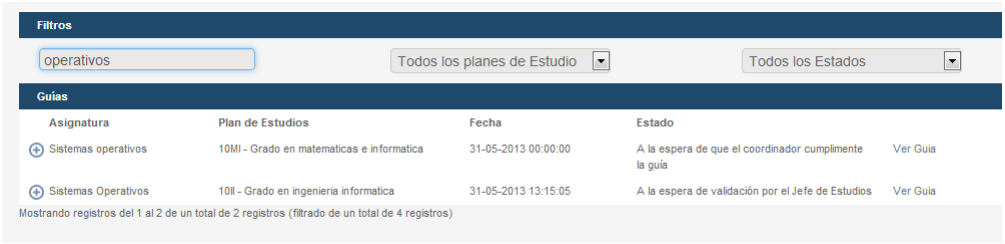

Figura 5.4: Ejemplo de tabla filtrando los resultados según su nombre

5.2.3. Problemática de la gestión dinámica de tablas

Por último, en una de las tablas se nos planteó el caso particular de querer obtener información adicional en función del contexto de la fila, que visualmente se mostrara dicha información como una extensión de la fila y que, además, la tabla no perdiera sus propiedades de filtrado y ordenación. Esta tarea, en principio compleja, no requirió más que el análisis detallado de la API del plugin DataTables.
El citado plugin posee una serie de funciones que permiten la inserción de un "div" auxiliar, en el que puede incrustarse cualquier clase de contenido. La funcionalidad, por tanto, quedó reducida a una llamada AJAX que obtuviera el citado contenido, y que en su resolución, abriera ese "div" auxiliar e incrustara el contenido recibido.

Figura 5.5: Ejemplo de tabla expandida

5.3. Conclusiones y futuras líneas de trabajo

Las conclusiones obtenidas tras el uso de estos sistemas de tablas dinámicas es que, gracias a la multitud de plugins existentes, son un recurso fácil de implementar y de mantener, siempre que existan las funcionalidades requeridas para la problemática planteada en cada caso.
Destacar como posible línea de investigación futura la conversión de la funcionalidad de expansión de las filas de una tabla a un plugin propio de DataTables, de manera que su inclusión sea inmediata y su utilización más simple.

Parte III

Herramientas de Desarrollo

Capítulo 6

Biblioteca de funcionalidades comunes en Javascript

Debido a la magnitud del proyecto al que nos enfrentábamos, se hizo patente desde el primer momento la necesidad de crear una biblioteca de funcionalidades comunes para todo el proyecto, dada la gran cantidad de código javascript que preveíamos crear. Dicha biblioteca pretendía contener toda una serie de funciones de uso común, tanto en el contexto de nuestro proyecto como para dotar al propio lenguaje de las utilidades que otros lenguajes ofrecen.

6.1. Análisis lenguaje Javascript

Profundizando más en este punto, Javascript es un lenguaje de programación cuyo paradigma hace que el desarrollador se vea obligado a replantear gran parte de la estructura y los algoritmos que conoce y domina con comodidad, y esto es debido, principalmente, a que es un lenguaje "orientado a eventos" y con "prototipos". La implicación más directa de esto es que no es "natural" estructurar el código a los patrones con objetos más clásicos, debido, principalmente, a que no posee una estructura clara de encapsulación de atributos; no existen modificadores explícitos tales como "public", "private" o "protected" que sí poseen otros lenguajes, ofreciendo solamente los mecanismos de ámbito de variables como manera eficaz de encapsulación.
No obstante, si puede simularse una suerte de métodos privados en un objeto, es decir, aquellos a los que no se puede acceder si no es dentro del propio objeto; tan solo habría que especificar en el constructor que dicho método no pertenece al ámbito "this".
Veamos un ejemplo:

```
holaMundo = function(nombre){
    // Atributo publico
    this.nombre = nombre;
    // Metodo publico
    this.saludar = function(){
        alert("Hola " + this.nombre);
    }
    // Metodo Privado
    function mundo(){
        alert("privado " + this.nombre);
    }
}
var obj = new holaMundo("mundo!");
obj.saludar(); // --> "Hola Mundo!"
obj.mundo(); // Error
```

Listado 6.1: Ejemplo de Método Privado

Tampoco existe un sistema de herencia propiamente dicho; en su lugar, toda instancia de un objeto posee el atributo "prototype", desde el que podemos crear un prototipo del objeto. Supongamos el siguiente fragmento de código:

```
var padre = function(nombre){
    this.nombre = nombre;
    this.quienSoy = function(){
        return this.nombre;
    }
}
var hijo = function(nombre){
    this.nombre = nombre;
    this.quienSoy2 = function(){
        return this.nombre;
    }
}
// Forzamos la herencia
hijo.prototype = new padre();
var obj = new hijo("juan");
alert("hijo: " + obj.quienSoy2() + " padre: " + obj.quienSoy());
// --> "hijo: juan padre: juan"
```

Listado 6.2: Ejemplo de herencia mediante prototipos

Como puede apreciarse en el código arriba expuesto, para realizar una herencia debemos, por una parte, crear dos objetos, padre e hijo, y luego asignar al prototipo del objeto hijo el objeto padre. De esta forma, conseguiremos heredar todos los métodos del objeto padre.

A la vista de esto, podríamos emular una herencia completa, es decir, tener una referencia al padre mientras que heredamos sus métodos y los sobrescribimos, de manera que tengamos acceso a toda la funcionalidad, pero para ello habría que realizar una copia completa del objeto padre de manera manual (el problema de la copia de objetos se tratará más tarde en este mismo capítulo).

Otra de las propiedades del prototipado de objetos es que podemos ampliar la funcionalidad de cualquier objeto con tan solo añadir la propiedad o método que queramos a

su prototipo; de esta manera, podríamos, por ejemplo, transformar el objeto array para añadir una funcionalidad que nos interesara, como, por ejemplo, la ordenación de un array de objetos en función de uno de sus valores concretos.

Por último, destacar que el lenguaje javascript también adolece de la carencia de espacios de nombres, aunque esta carencia puede subsanarse mediante el uso de objetos, definiendo un objeto envolvente que contendrá todas las funciones, atributos y objetos del espacio de nombres.

En resumen, y a la vista de lo expuesto, el lenguaje tiene una serie de carencias fácilmente subsanables, y esa es, en gran medida, la motivación de la biblioteca de funciones que desde un primer momento se decidió crear; además, se han añadido a dicha biblioteca toda una serie de funciones que, si bien no son apoyo del lenguaje, si se trata de una serie de rutinas, más centradas en el contexto de nuestra plataforma, y que hemos decidido encapsular para mejorar su legibilidad y depuración de los errores que puedan ocasionar.

6.2. Funciones de apoyo al lenguaje

Para facilitar el proceso de adaptación del desarrollador al lenguaje, así como para facilitar la escalabilidad, encapsulado y modularidad del código, se proponen las siguientes funciones:

- **clone:** Se trata de una función que se incluirá en el prototipo del objeto "Object" y que, por tanto, será heredada por todos los objetos. Tendrá la capacidad de crear una nueva instancia de un objeto, sin tener ninguna conexión con el primero.

```
1  Object.prototype.clone = function(){
2      var newObj = (this instanceof Array) ? [] : {};
3      for (var i in this) {
4          if (this[i] && typeof this[i] == "object") {
5              newObj[i] = this[i].clone();
6          }
7          else
8          {
9              newObj[i] = this[i];
10         }
11     }
12     return newObj;
13 }
14
15 // funcionamiento
16
17 holaMundo = function(nombre){
18     // Atributo publico
19     this.nombre = nombre;
20     // Metodo publico
21     this.saludar = function(){
22         alert("Hola " + this.nombre);
23     }
24     // Metodo Privado
25     function mundo(){
```

```
26          alert("privado " + this.nombre);
27      }
28  }
29  var obj = new holaMundo("mundo!");
30  obj2 = obj.clone();
31  obj2.nombre = "pepito!";
32  obj.saludar(); // --> "Hola Mundo!"
33  obj2.saludar(); // --> "hola pepito!""
```

Listado 6.3: Definición y comportamiento de la función "clon"

- **namespaces:** Se trata de una función capaz de generar espacios de nombres (basados en objetos) de la forma "espacio.subespacio.subespacio". Es capaz de reconocer los espacios de nombres ya existentes y concatenar los nuevos subespacios de la forma adecuada. La función devuelve una referencia al ultimo espacio de nombres.

```
1  function namespace(namespaceString) {
2      var parts = namespaceString.split('.'),
3          parent = window,
4          currentPart = '';
5
6      for(var i = 0, length = parts.length; i < length; i++) {
7          currentPart = parts[i];
8          parent[currentPart] = parent[currentPart] || {};
9          parent = parent[currentPart];
10     }
11
12     return parent;
13 }
14
15 namespace("utils");
16 namespace("utils.math");
17 var suma = namespace("utils.math.sum");
```

Listado 6.4: Definición y comportamiento de la función "namespace"

- **Valor por defecto:** Se trata de una función que asigna un valor por defecto a una variable en el caso de que dicha variable no se haya definido.

```
1      function defaultValue(arg, def) {
2          // Si el contenido no esta definido,
3          // lo cambia a un valor por defecto
4          return (typeof arg == 'undefined' ? def : arg);
5      }
```

Listado 6.5: Definición de la función "'defaultValue"

6.3. Funciones específicas de la plataforma

En esta sección mencionaremos otras funciones añadidas a la biblioteca, cuyo cometido es, principalmente la reutilización de código a través de la plataforma, ofreciendo ciertas utilidades relacionadas con los distintos aspectos de la misma.

- **Funciones de cadenas de texto:** Destinadas a procesar cadenas de texto para adecuarlas a las necesidades de la plataforma.

  - *toTitleCase(str)* cuya finalidad es ofrecer una manera sencilla de establecer la primera letra de la cadena de texto dada en mayúsculas y, el resto, en minúsculas.
  - *htmlentitiesToUTF8(str)*, que surge de la necesidad de transformar entidades html a codificación UTF-8, debido a que la función *json_encode* del lenguaje PHP presenta problemas con determinadas codificaciones de caracteres y, en vista de ello, se decidió que toda información que fuera a ser tratada por esa función se codificara primero como entidades HTML (que utilizan codificación ASCII) y, una vez dichos elementos estuvieran en el lado del cliente, se decodificarían utilizando la función *toTitleCase(str)*.

- **Funciones de Validación:** Se trata de rutinas que aportan determinadas facilidades al complejo proceso de validación de campos.

  - *validateError(selector, div, errorValue)*, que determina si el valor del selector dado coincide con el valor considerado erróneo; de ser así, añade al div contenedor de dicho selector la clase error de Bootstrap, y si no, elimina dicha clase, en caso de que la tuviera.
  - *validateOnFocusOut(selector, regExp, matchOrNot, length, callback)*, capaz de validar el valor contenido en el selector, haciendo que coincida (o no) con una expresión regular dada e imponiendo que la longitud de dicho valor sea menor que la dada; de no cumplirse todas estas condiciones al disparar el evento "focusout", el campo se mostrará como erróneo, añadiendo la clase de error de Bootstrap y un mensaje aclaratorio sobre el tipo de error producido. Adicionalmente, se permite el paso de un callback, que se ejecutará tras dispararse el evento (sea o no válido el campo).
  - *InitTooltips()*, que inicializa todos los tooltips, es decir, pequeñas capas emergentes que aportan información útil para el usuario cuando pasa el puntero del ratón sobre dicho campo, disparando el evento "hover".
  - *cleanForm(id, event, callback)*, que elimina las clases de error y establece el contenido de los campos a su valor original para todos los campos marcados con la clase "cleanable".

- **Funciones para la interactuación con la consola de estado:** Para facilitar información sobre el estado actual de las acciones realizadas por los usuarios, así como para informar en cada momento de la ultima acción realizada, se ha implementado una consola, implementada como un div flotante, que está presente en gran parte de la aplicación. Para facilitar el trabajo de escritura sobre dicha consola, se han creado las siguientes funciones:

  - *getFecha()* y *getHora()*, que ofrecen una representación legible de la fecha y hora actual.
  - *animateConsola()*, que crea sobre la consola una pequeña animación de rebotes, haciendo que el usuario centre la vista en ella.

  - *consoleMsg(icon, color, msg)*, que escribe sobre la consola el mensaje dado, precedido de la fecha y hora actuales y el icono indicado (tic de confirmación, cruz de error y una espiral animada para indicar procesos en espera).
  - *consoleImpMsg(icon, selector, msg)*, que escribe sobre el selector dado, el mensaje pasado como argumento, precedido de la hora y el símbolo especificado.

- **Funciones para control de suciedad en los campos de texto:** Con el fin de evitar que la funcionalidad de autoguardado al perder foco generara demasiadas llamadas a la hora de escribir en un campo de texto, se han implementado estas funciones, encargadas de comprobar el nivel de "suciedad" del campo de texto.

  - *initDirty()*, que inicializa, sobre los elementos con la clase "trackable", el medidor de suciedad.
  - *isDirty()* y *isDirty(input)*, que devuelven true si el valor respecto al ultimo guardado ha cambiado, para todos los campos "trackable" en el primer caso y solo para el input específico en el segundo.

- **Funciones misceláneas:** Englobamos aquí toda una serie de funciones de propósitos diversos:

  - *sort_unique(arr):* Devuelve la versión ordenada y sin elementos repetidos del array dado.
  - *inArraySelectableObjs(obj, array):* Devuelve la posición del objeto dado en el array dado si la propiedad "label" de ambos coincide; en caso de no hacerlo, devuelve -1.
  - *forceSelected(selector, val):* Fuerza a que, en el espacio de nombres indicado por "selector" que pertenece a un autocompletado, se establezca como "selected" el objeto cuyo label coincide con el valor pasado.
  - *abrir_menu()* y *cerrar_menu():* Ofrece la funcionalidad para mostrar y ocultar el menú lateral derecho de la plataforma.

6.4. Conclusiones y futuras líneas de trabajo

Existen muchas líneas de trabajo que puede aportar, tanto al lenguaje como a la plataforma, una riqueza y funcionalidad que ayudarían en gran medida a los futuros desarrolladores. Por mencionar uno de los problemas más clásicos y sobre los que más artículos se han escrito, se propone tratar de ofrecer una funcionalidad que emule de manera completa los sistemas de herencia de otros lenguajes de programación orientados a objetos, es decir, dando no solo la herencia de métodos (cosa que sí se puede conseguir fácilmente mediante el uso de prototype)sino también aportando un método eficaz de acceder a los métodos del objeto del que se hereda (mediante, por ejemplo, el acceso a un atributo "super").

Capítulo 7

Herramientas Auxiliares para el Desarrollo

En este capítulo, se aborda cuales fueron las herramientas auxiliares desarrolladas, así como la problemática que los originó, para volver más sencilla la tarea del desarrollo, depuración y acceso a los datos almacenados en las distintas bases de datos.

7.1. Herramienta de Generación de Código

7.1.1. Motivación y análisis de la solución

A la hora de plantear un posible mantenimiento, además de la inclusión de nuevo código a la plataforma, por personas ajenas a nuestro equipo, se decidió que, debido a la complejidad de determinadas partes de la plataforma, se crearía una pequeña utilidad capaz de generar esqueletos en los distintos lenguajes de programación para facilitar la estructura del código, así como para que éste esté comentado y el desarrollador tenga la comodidad de trabajar sobre una pieza de código parcialmente funcional.

En nuestro caso, la herramienta debía ser capaz de proporcionar flexibilidad a la hora de ofrecer plantillas, tanto ya creadas por el actual equipo como añadidas posteriormente, de forma rápida, clara y práctica; además, el sistema de plantillas debía ser lo más cómodo posible para el usuario, ofreciendo funcionalidades como creación de tablas, de llamadas AJAX, de formularios completos de validación, etc. de una manera simple y escalable.

Por todo ello, decidimos crear la herramienta utilizando el motor de plantillas "Smarty", obteniendo, por una parte, toda la potencia que ofrece dicho motor (inclusión de plantillas, bucles, saltos condicionales, sustitución de variables, etc.) y por otra garantizando la integración con el resto de la plataforma, ya que el sistema está creado para soportar el uso de este motor de plantillas.

7.1.2. Construcción de la herramienta

Una vez decidido el motor y el soporte de la herramienta, queda la tarea de codificar la herramienta. El código es el siguiente:

```
1  <?php
2
3  /*
4  Existe una variable $con que esta asociada al
5  template a imprimir.
6  */
7
8  if(isset($_POST['tpl_name'])){
9      $codeVars = array_slice($_POST, 1);
10     foreach ($codeVars as $key => $value) {
11         $con->assing($key,$value);
12     }
13     $con->display($_POST['tpl_name']);
14 }
```

Listado 7.1: Herramienta de generación de código

7.2. Sistema de Consultas a la Base de Datos

Uno de los problemas surgidos durante el desarrollo de la plataforma fue el acceso a la información almacenada en la base de datos; la raíz de esta problemática se encuentra en la incompatibilidad (por motivos que aún desconocemos) de ciertas máquinas del laboratorio con el programa utilizado para realizar las consultas sobre la base de datos: Toad for Oracle.

Tras probar varias versiones y comprobar que ninguna de ellas funcionaba correctamente, llegando a no comenzar a ejecutar nunca, se tomó la decisión de crear un pequeño sistema capaz de ejecutar querys SQL utilizando el lenguaje PHP.

Al tener la plataforma acceso directo a la base de datos, un sistema de este tipo nos permite utilizar toda la flexibilidad del lenguaje SQL tan solo recurriendo a la clase ADOdb, que encapsula los pormenores de la configuración de las llamadas a la base de datos, ofreciendo un interfaz directo sobre el que ejecutar los mandatos. Una vez obtenido un objeto de dicha clase, el sistema toma la consulta directamente de un area de texto y lo ejecuta; el resultado se muestra justo debajo del citado cuadro de texto, informando sobre la consulta lanzada, contra que base de datos se realizó dicha consulta y cuanto tiempo se ha tardado en ejecutar, además de mostrar los resultados de forma legible, aunque no en forma de tabla.

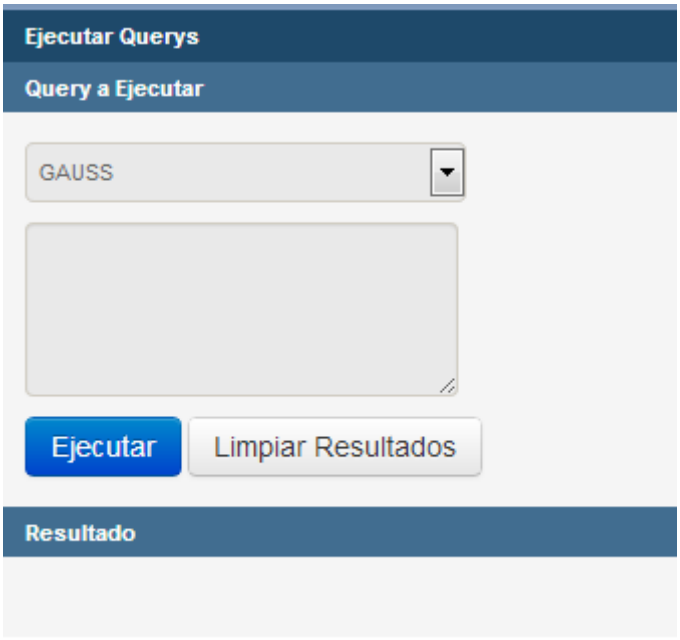

Figura 7.1: Sistema de ejecución de querys

Figura 7.2: Ejemplo de ejecución de una query

Este sistema está solamente planteado para facilitar la labor de los desarrolladores, y de ninguna manera debe ser incluido en la versión puesta en producción; para evitar la confusión, al realizar el paso a producción se eliminarán los dos ficheros que componen esta herramienta.

7.3. Herramientas de ayuda a la Depuración de Código

En esta sección englobamos varios fragmentos de código utilizados, tanto en javascript como en PHP, para, de una manera u otra, obtener información acerca del estado actual de las variables.

7.3.1. Depuración en PHP: *printjs()* y *getSessionVar()*

Para conseguir obtener los datos de las variables manejadas por PHP sin que estorbaran visualmente, se decidió crear estas funciones que, tras ser llamadas, crearían mensajes u objetos javascript con la información PHP, pero sin crear "ruido" en el resto de la plataforma.
La máxima de no interferir visualmente se debe a que muchas veces era necesario que dos personas trabajaran sobre funcionalidades presentes en una misma pantalla, y podría darse el caso de interferir el uno en el trabajo del otro; de esta forma, se mantiene una clara distinción entre la funcionalidad y las trazas de depuración, quedando estas últimas relegadas a la consola javascript (la cual, por defecto, está oculta). Se plantearon otros métodos de depuración, como usar sistemas de logs de bibliotecas de terceros o escribir la traza de depuración sobre un fichero de texto, pero ambas fueron tempranamente descartadas, la primera por generar los mismos problemas que la simple depuración mediante mensajes y la segunda para no crear un fichero de log de gran tamaño.

Por tanto, se tomó la decisión de crear dos funciones muy específicas: una primera, *printjs()*, de caracter muy general, capaz de mostrar mensajes simples o el valor de una variable u objeto, haciendo uso de la sentencia de javascript *console.log()*, que está disponible en la gran mayoría de navegadores actuales.

```
1  <?php
2
3  function printjs($msg, $obj=""){
4      echo '<script type="text/javascript">';
5      echo 'console.log('.json_encode($msg).');';
6      if (strcmp($obj, "")){
7          echo $obj . '='.json_encode($msg).';';
8      }
9      echo '</script>';
10 }
```

Listado 7.2: Definición de la función printjs()

La segunda función, *getSessionVar()*, toma los valores almacenados en "$_SESSION" y los almcena en un array javacript llamado "_session". De esta forma, se pueden consultar valores concretos de la sesión con tan solo acceder a ellos mediante su índice.

```
1  <?php
2
```

```
3 function getSessionVar(){
4     echo '<script type="text/javascript">';
5     echo '_session = '.json_encode($_SESSION).';';
6     echo '</script>';
7 }
```

Listado 7.3: Definición de la función getSessionVar()

Como se puede observar, ambas funciones son muy simples y, no obstante, cumplen a la perfección su cometido, motivo por el que decidimos utilizarlas, ya que su tiempo de codificación es ínfimo y su utilidad, incalculable.

7.3.2. Depuración en javascript: encapsulación del objeto *console*

La depuración en javascript se vuelve más compleja que su homóloga en PHP, debido a que no tiene mucho sentido interrumpir el flujo de ejecución de un script en éste lenguaje, siendo como es un lenguaje orientado a eventos (y por tanto, sin una traza de ejecución definida a priori). Por tanto se decidió también desde una fase bastante temprana la depuración por medio de mensajes en la consola.
A la hora de investigar esta posibilidad, nos encontramos con la problemática de que, si bien el objeto *console* existe para casi todos los navegadores actuales, no todos ellos ofrecen las funcionalidades que requeríamos; entre ellas, se nos hacía imprescindible el par de funciones *console.group()* y *console.groupEnd()*, las cuales agrupan los subsiguientes mensajes impresos, ya sea por medio de la función *console.log()* como por cualquiera de sus derivados, bajo un mismo ámbito, de manera que puede localizarse rápidamente el valor que estamos buscando.

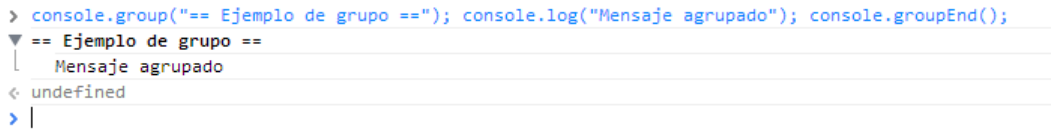

Figura 7.3: Ejemplo de las funciones *console.group()* y *console.groupEnd()*

Además, necesitábamos una manera eficaz de desactivar todos los mensajes de log mediante una única variable; para ello se nos plantearon dos alternativas:

- Sobreescribir todas las funciones del objeto *console* a funciones vacías, de manera que no se generara un error en el código al ser llamadas pero que no imprimieran nada.
- Crear una serie de funciones envolventes que, en función de conmutador (una variable global) permitieran, o no, la impresión de mensajes.

Uniendo las os problemáticas arriba descritas, llegamos a la conclusión de que lo más factible era la creación de un nuevo objeto que, por una parte, comprobara la existencia de las funciones que necesitábamos y, en caso de no existir, emularlas y, por otro, que fuera capaz de activar y desactivar la impresión de mensajes. El código resultante se muestra a continuación:

```
1 var debugJs = function(){
2     // Funcion vacia
3     var noop = function(){}
```

```
4
5       // Variable que controla la escritura de mensajes
6       this.DEBUG = true;
7       // Variable que controla el grupo actual
8       this.GROUP = '';
9       // Comprobamos la existencia del objeto console
10      if (typeof console != "undefined") {
11          // Creamos la funcion log
12          this.log = function(obj){
13              if(this.DEBUG){
14                  if(typeof console.group != "undefined"){
15                      console.log(obj);
16                  } else {
17                      console.log("["+this.GROUP+"] " + obj);
18                  }
19              }
20          }
21
22          // creamos la funcion auxiliar group, comprobando si existe
23          var _auxGroup = console.group || function(obj){this.group = obj;}
24          // creamos la funcion auxiliar groupEnd, comprobando si existe
25          var _auxGroupEnd = console.groupEnd || function(){this.group = '';}
26
27          //Creamos las funciones envolventes
28          this.group = function(obj){
29              if(this.DEBUG){
30                  _auxGroup.call(console, obj);
31              }
32          }
33          this.groupEnd = function(obj){
34              if(this.DEBUG){
35                  _auxGroupEnd.call(console);
36              }
37          }
38      } else {
39          this.log = noop;
40          this.group = noop;
41          this.groupEnd = noop;
42      }
43  }
```

Listado 7.4: Definición de la función debugJs

7.4. Conclusiones y Futuras Líneas de Trabajo

Como conclusión, muchas son las posibles herramientas que pueden diseñarse para facilitar la codificación de una plataforma de las dimensiones de la que nos ocupa, pero al tratarse de rutinas tan específicas las que necesitan ser codificadas que es una decisión acertada realizar dichas herramientas en función de las necesidades del momento.

Como futura línea de investigación, se propone mejorar el sistema de depuración para javascript, haciéndolo compatible en la medida de lo posible con cualquier navegador, e incluso desarrollar una consola de depuración completamente independiente de la plataforma.

Parte IV

Conclusiones

Capítulo 8

Resultados obtenidos

Una vez examinados tanto los algoritmos como las herramientas utilizadas, pasaremos a comentar, someramente, los resultados obtenidos a la hora de realizar la plataforma.

El primer apunte positivo a resaltar es que, gracias a la potencia que ofrece la tecnología de llamadas asíncronas hemos podido organizar la plataforma de manera lógica y escalable, permitiendo incluso que determinadas partes del apartado de cumplimentación de la guía docente puedan reordenarse o incluso, ocultarse; para ello, hemos basado todo el sistema de cumplimentación en una serie de llamadas AJAX a los ficheros que contienen la funcionalidad, de manera que éstos se cargan de forma dinámica.

Figura 8.1: Visualización del menú dinámico y la imagen de espera entre la carga de distintas secciones

Además, la tecnología de llamadas asíncronas también nos ha permitido implementar un sistema de autoguardado, que se disparará cuando el usuario pierda el foco de un campo susceptible de ser guardad. De esta forma, se evita que, en caso de perder la conexión con la plataforma por cualquier motivo, se minimice la pérdida de datos

Otro punto a destacar es el aspecto y funcionalidad de los formularios, capaces de detectar e informar adecuadamente tanto de los errores en el formato como de las posibles soluciones a los mismos.

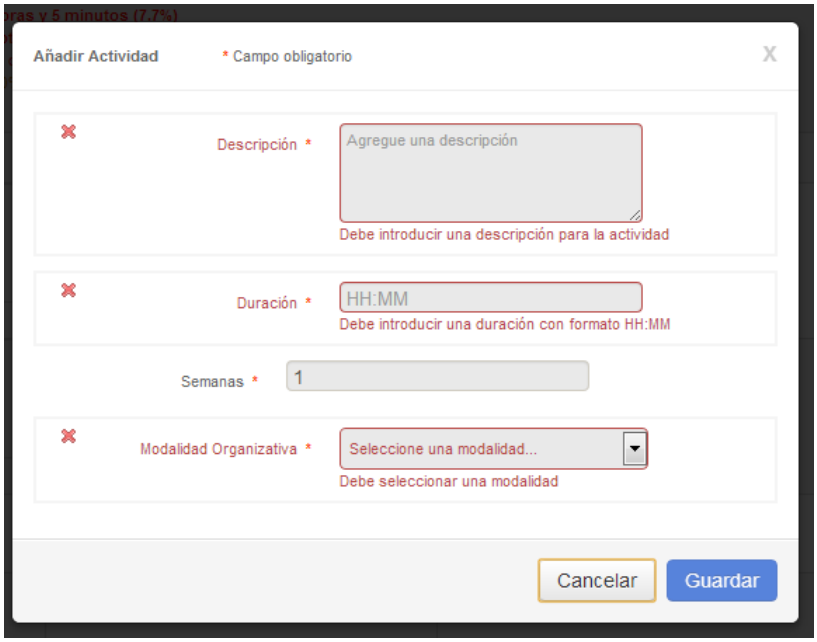

Figura 8.2: Ejemplo de un formulario con campos erróneos y botón deshabilitado

También es necesario comentar que el aspecto final de los autocompletados y las tablas reordenables, que ofrecen una forma rápida tanto de introducir datos como de consultarlos.

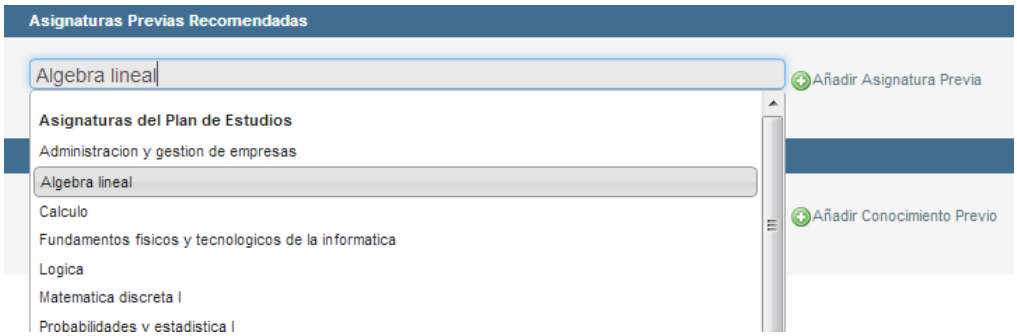

Figura 8.3: Aspecto final del sistema de autocompletado

Figura 8.4: Aspecto final de una tabla reordenable

También cabe destacar la herramienta de importación de guías docentes, desde la que pueden importarse tanto guías completas como sólo fragmentos de éstas.

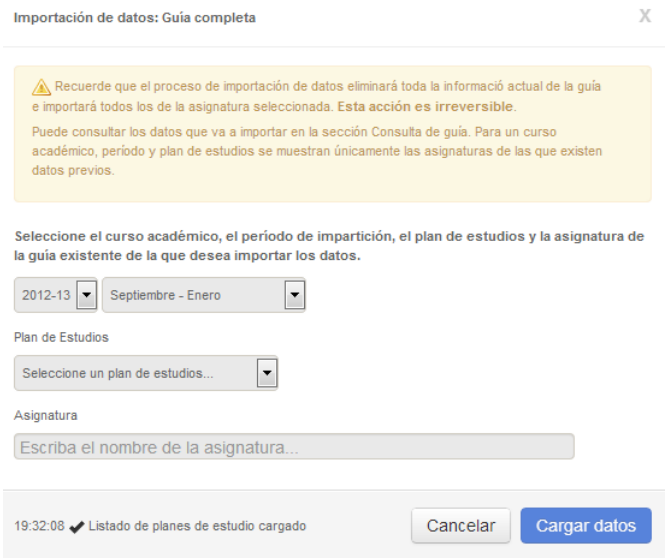

Figura 8.5: Formulario para la importación de datos

Todos estos componentes crean una plataforma ágil y fácilmente usable, cuyo propósito, facilitar el proceso de cumplimentación y aprobación de guías docentes, queda alcanzado de una forma satisfactoria.

Capítulo 9

Conclusiones

Para concluir, me gustaría realizar una reflexión acerca del trabajo realizado y algunas futuras líneas de trabajo futuro.
En primer lugar, destacar lo complicado que puede resultar la realización de un proyecto enfocado desde el punto de vista del usuario, ya que, técnicamente hablando, puede resultar muy complicado realizar un diseño simple e intuitivo. Por ello puedo decir que mi equipo y yo nos sentimos muy orgullosos del trabajo realizado y que, hasta ahora, las criticas recibidas son bastante buenas.

A la hora de abordar los puntos más conflictivos que hemos tenido que solventar, no puedo dejar de destacar el sistema de clases utilizado en la plataforma, que, si bien no estaba mal planteado en un inicio, operaba a un nivel demasiado bajo, no aportando la abstracción necesaria en algunos puntos; además, no ofrecía ningún tipo de ayuda a la hora de realizar consultas sobre claves ajenas en las tablas, delegando dicha funcionalidad en el código y ralentizando su desarrollo, al mismo tiempo que incrementaba la complejidad del código. A este punto hay que añadir que la estructura de la que partíamos se basa en un sistema de reenvío de formularios y, por tanto, si un usuario decide utilizar el botón "atrás" de su navegador, se encontrará con un molesto mensaje de reenvío de formulario, motivo por el que, desde un principio, abordamos la plataforma desde la perspectiva de implementar un sistema de llamadas AJAX, de manera que pudiéramos solventar este escollo, permitiendo al usuario navegar por la plataforma sin necesidad de utilizar el citado botón. Por último, destacar como punto conflictivo la poca flexibilidad de la que disponíamos para modificar el árbol de directorios, ya que éste estaba construido de manera que fuera compatible con el resto de utilidades de la plataforma, y la imposibilidad de crear entradas de menú a nuestro antojo, ya que para hacerlo se requería una nueva entrada en una tabla a la que no teníamos acceso.

Además, debido a la imposición en las fechas de entrega y por falta de tiempo no hemos podido refactorizar todas las partes del código ni realizar una buena documentación del mismo. Por lo tanto, se propone como futura línea de trabajo general homogeneizar todo el código para establecer una única línea de codificación, estableciendo los algoritmos explicados en este trabajo en todas las zonas del código, al tiempo que se comenta adecuadamente y se elimina el código redundante. Se propone también revisar y reimplementar todo el sistema de clases para que, sin realizar modificaciones sobre el modelo de datos existente, se creen unas clases funcionales y que ofrezcan un nivel de abstracción suficiente como para independizar el modelo de tablas de la aplicación en sí. Por último, se propone el estudio del impacto que supondría la migración del código a un soporte ba-

sado en algún framework de terceros que ofrezca más herramientas para la comodidad del desarrollo y su mejor mantenimiento. Una vez realizado dicho estudio, y si resulta viable dicha migración, comenzar a investigar qué framework de los muchos disponibles sería el más idóneo para albergar este proyecto y los que vengan en el futuro, centrándonos en temas tan importantes como la seguridad (evitando ataques de tipo *XSS*) o la reutilización de código (utilizando los sistemas de importación y plantillas que ofrecen casi todos los sistemas actuales de este tipo). Ésto, unido a la utilización de un sistema de Mapeo Objeto-Relacional para el modelo de datos dotaría a todo el sistema de una robustez y simplicidad mayores de las que ahora posee.

Bibliografía

[1] **Javascript a Boli**, *Evaluación perezosa de operadores AND y OR (lazy evaluation)*
http://notasjs.blogspot.com.es/2013/02/evaluacion-perezosa-de-operadores-y.html

[2] **PHP Web-Site**,
http://www.php.net

[3] **elegantcode**, *Basic Javascript part 8: Namespaces*
http://elegantcode.com/2011/01/26/basic-javascript-part-8-namespaces/

[4] **StackOverflow forums**, *Most efficient way to clone an object?*
http://stackoverflow.com/questions/122102/most-efficient-way-to-clone-an-object

[5] **BeOSmAn's Blooog**, *Clonar un objeto en Javascript*
http://beosman.org/archivo/2009/informatica/clonar-un-objeto-en-javascript.html

[6] **etnassoft**, *El valor de "this" en javascript: Cómo manejarlo correctamente*
http://www.etnassoft.com/2012/01/12/el-valor-de-this-en-javascript-como-manejarlo-correctamente/

[7] **jQuery**,
http://jquery.com/

[8] **jQuery UI**,
http://jqueryui.com/

[9] **Douglas Crockford**, *Classical Inheritance in JavaScript*
http://javascript.crockford.com/inheritance.html

[10] **Douglas Crockford**, *Prototypal Inheritance in JavaScript*
http://javascript.crockford.com/prototypal.html

[11] **Douglas Crockford**, *Private Members in JavaScript*
http://javascript.crockford.com/private.html

[12] **nestedSortable jQuery Plugin**,
http://mjsarfatti.com/sandbox/nestedSortable/

[13] **DataTables Jquery Plugin**,
http://datatables.net/

[14] **DataTables Column Filter Add-on**,
http://jquery-datatables-column-filter.googlecode.com/svn/trunk/index.html

[15] **Chris Schnaars**, *Building jQuery DataTables with expandable accordion rows*
http://www.chrisschnaars.org/2011/11/16/building-jquery-datatables-with-expandable-accordion-rows/

[16] **jQuery UI Autocomplete**,
http://jqueryui.com/autocomplete

[17] **Bootstrap**,
http://twitter.github.io/bootstrap/

[18] **FlexBox Autocomplete**,
http://www.fairwaytech.com/flexbox/

[19] **Scriptaculous Autocomplete**,
http://script.aculo.us/

[20] **Creativos Online**, *30 Scripts de autocompletar*
http://www.creativosonline.org/blog/30-scripts-de-autocompletar.html

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Fri Feb 14 19:20:16 CET 2014
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)